Rigorous Process - 2024 Summer

Validation is the proof, that a system meets its requirements.

Verification is the proof, that the program is working mathematically correct.

Design by Contract

Every publicly accessible method defines pre- and postconditions. The client calls a method, the supplier provides the method.

> **IF** a client runs the method in situations that satisfy the precondtion - **THEN** the supplier will make sure that the method execution delivers a state that satisfies the postcondition

Class context

Class invariant (assertion involving the instance variables) must be satisfied at all times.

- After the constructor execution
- Before and after execution of every public method

> Private methods can violate the invariant!

Inheritance

When a class gets extended, the overridden methods cannot demand more but must at least provide the same. The client only knows the general implementation of the superclass, so they need to be able to assume the same outcome when satisfying the precondition.

This means:

- The **preconditions** of an overridden method must be the **same or weaker** than the super methods
- The **postconditions** of an overridden method must be the **same or stronger** than the super methods
- The **class invariant** can be the **same or stronger** as the superclass ones

Hoare Logic

Hoare Triple:

- Precondition ϕ
- Program P
- Postcondition ψ

> If program P is started in a state satisfying ϕ and if it terminates, then the finish state will stisfy ψ

Procedure:

- down-to-top: Postcondition must lead to precondition when executing all statements
- in loops:
  - **loop Guard** (Abbruchbedingung) must be true in loop, after loop ¬ guard must be true
  - **loop Invariant** is true:
    - before loop
    - begin of every loop execution
    - end of every loop execution
    - **only true when in loop**
- in if/else: every branch has a guard that holds in the branch itself

> For **total correctness** termination has to be proven !

Model checking

Advantages:

- general verification approach suitable for many different use cases
- supports partial verification → focus on essential properties first
- rapidly increase by interest in industry
- can easily be integrated in development cycles & therefore lead to shorter development times

Weaknesses:

- only checks a systems model, not the system itself → verification can only be as good as the model
- only suitable for control-heavy systems, not data-heavy systems
- checks only stated requirements → no guarantee of completeness
- state-space-explosion problem: number of states can easily exceed available space

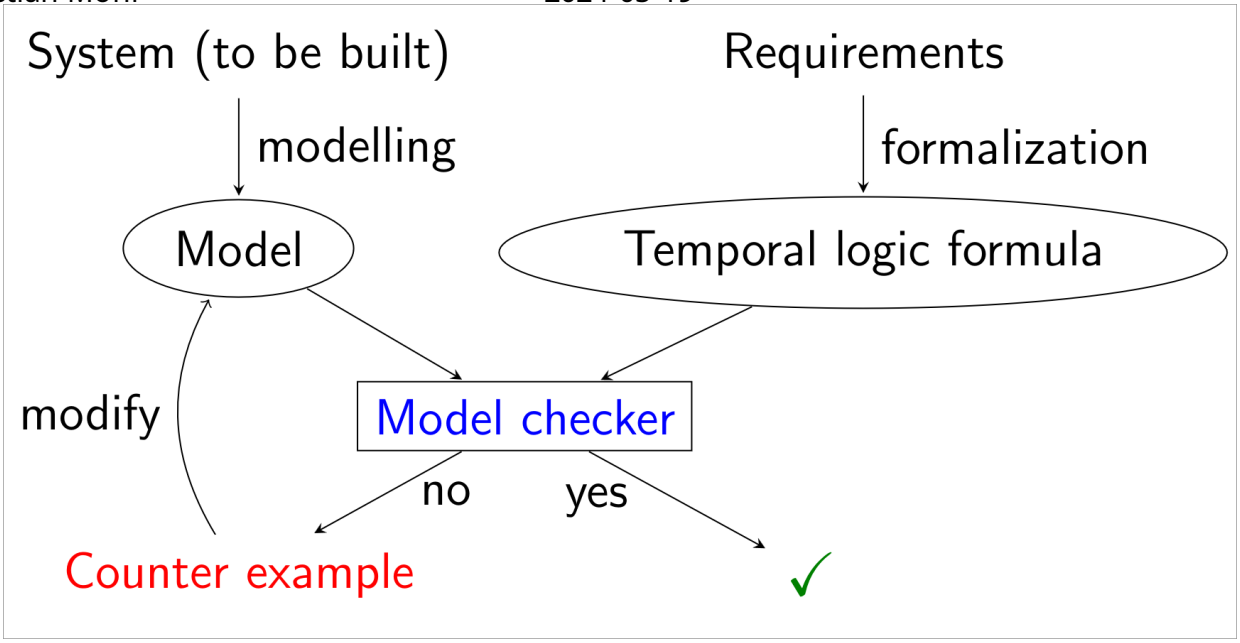
Kripke structure

A Kripke structure consists of a 4-tuple:

$$M = (S, S_0, R, L)$$

- S is a set of states
- $S_0 \subseteq S$ is a set of the initial states
- $R \subseteq S \times S$ are the transitions allowed for each state
- L gives the set of propositions allowed in each state

Higher-level language models (e.g. Promela) can be translated to Kripke structures.



LTL - Linear Temporal Logic

Connection of atomic propositions through boolean and temporal operators.

- Boolean connectives:
  - NOT: ¬
  - AND: ∧
  - OR: ∨
  - Implies: $\Longrightarrow$
  - Equivalence: $\Longleftrightarrow$
- Temporal operators:
  - Next: X = holds in next state
  - Globally / Always: G = holds always
  - Finally / Eventually: F = holds sometime in the future
  - Until: U = x holds until y holds, y will eventually hold

> Generally defines a set of runs of a concurrent system, rather than a single instance of a run.

CTL - Computational Temporal Logic

Connection of temporal operators with path quantifiers:

- Path quantifier:
  - On all paths: A
  - Exists one path: E

Path Notation:

$$model, starting\ state \models \rho$$
$$\rho = CTL\ formula$$
$$f.e.:\ M, s_0 \models E\ F\ p$$

This means, that in the *model* (Kripke strucutre) M *starting from state* s_0 ($s_0 \in M$) there exists (E) a path, that will eventually (F) accept p

Fairness in Model checking

Fairness means input requests are not ignored and messages continuously transmitted ar not never received.

Weak:

$$F\ G\ enabled(x) \Rightarrow G\ F\ executed(x)$$

Strong:

$$G\ F\ enabled(x) \Rightarrow G\ F\ executed(x)$$

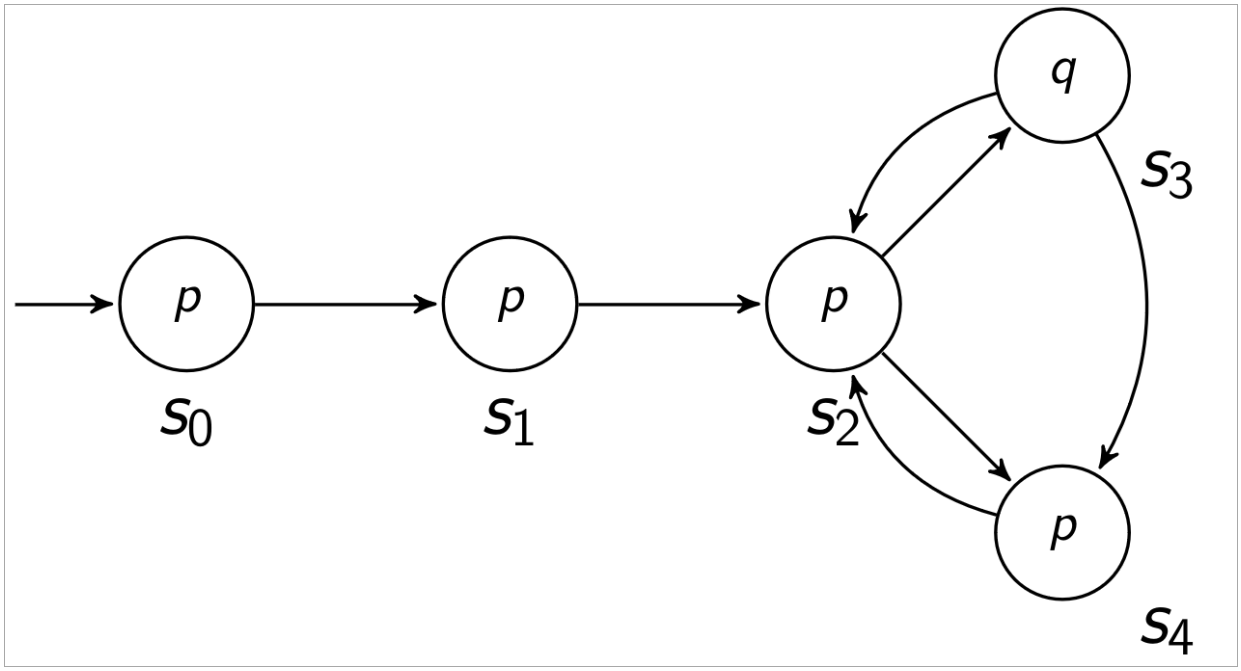
Weak fairness

Every statement, which is always enabled *from a certain point on*, is eventually executed.

Strong fairness

Every statement, which is *infinitely* often enabled, is eventually executed.

Example

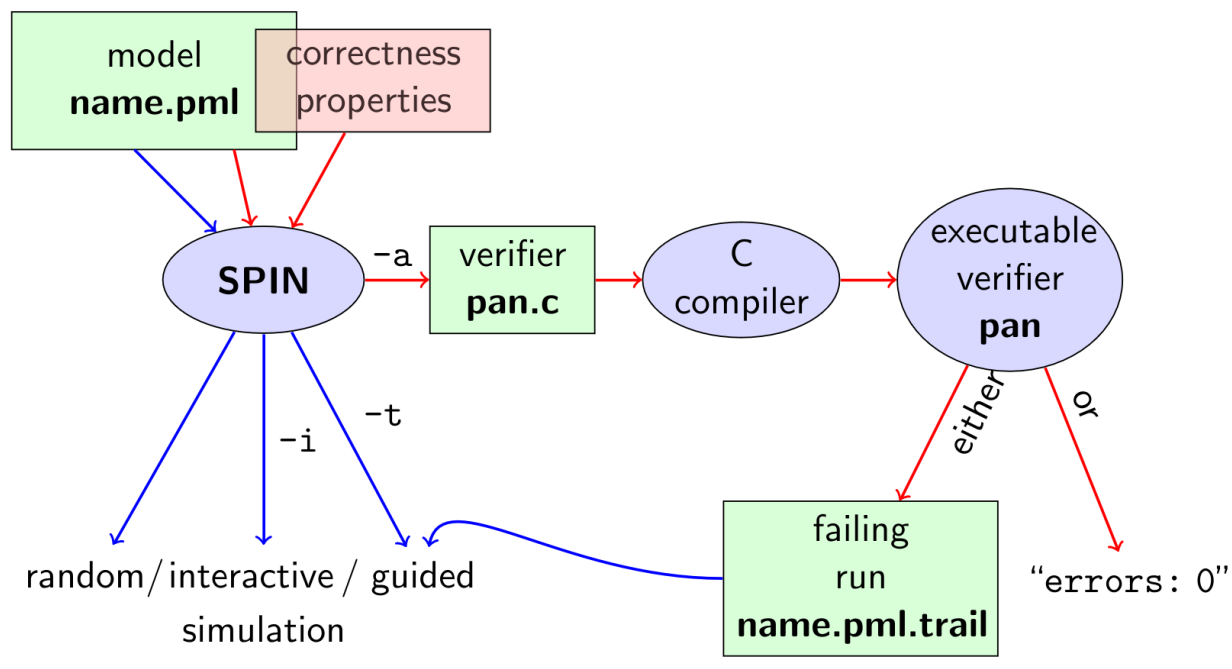


CTL: $\rho = A\,G(p \implies A\,F\,q)$

But the path is not fair, as it could be $M, s_0 \models \rho$ with the path being $s_0 s_1 (s_2 s_4)^\omega$, which would mean that s_3 never gets visited.

The fairness constraint to add would be $\mathrm{F} = \{\{s_3\}, \{s_4\}\} \rightarrow M^{'}$, which would leave to the fair path notation $M^{'}, s_0 \models_f \rho$.

Spin



Safety vs Lifeness in SPIN

Safety tries to prove that *nothing bad happens*, by proving that the system doesn't reach a bad state. Safety is proven by assertions.

```
ltl atMostOne { [] (critical <= 1) }
```

Lifeness tries to prove that *something good happens eventually*, by proving that the system does eventually reach a desired state. Lifeness is proven by LTL formulas.

```
ltl pWillEnterC { <> c}
```

SAT / SMT Solvers